Applying Model-Driven Engineering to High-Performance Computing:
Experience Report, Lessons Learned, and Remaining Challenges

Benoît Lelandais[a], Marie-Pierre Oudot[a], Benoît Combemale[b]

[a] *CEA, DAM, DIF, France*
[b] *University of Toulouse & Inria, France*

Abstract

Thanks to the increasing power of supercomputers, CEA develops ever more complex numerical simulators in the field of High Performance Computing (HPC) to cover a wide range of physical phenomena. As a consequence, simulation codes tend to become unmanageable and difficult to maintain and adapt to new hardware architectures.

In this paper, we report on our experience in the use of Model-Driven Engineering (MDE) and Domain-Specific Languages (DSLs) to face these challenges through two projects, namely Modane and NabLab. From this experience, we discuss the main lessons learned to be considered for conducting future projects in the field of HPC, and the remaining challenges that are worth being included in the road-map of the MDE community.

Keywords: Model-Driven Engineering, Modeling Language, Domain-Specific Language, Language Workbench, High-Performance Computing.

1. Introduction

CEA, the French Atomic Energy Commission, started in the late 1990s a supercomputing program based on successive high-performance computers. These computers are used to run parallel numerical simulation codes, simulating physical phenomena (*e.g.*, laser experiments). These simulators classically manage 2D or 3D meshes composed of millions of elements to represent the geometric and numerical context of the application.

Thanks to the increasing power of supercomputers, a wider range of physical phenomena can be covered. As a result, the size and complexity of numerical simulators continue to grow. Keeping them in production, making them evolve, and maintaining the quality level become a real challenge. This led to the development, in collaboration with IFPEN[1], of the Arcane C++ framework [1] to handle technical aspects of simulators such as mesh, memory management, I/O and parallelism. The development of Arcane started in 2000, and the framework is now used by several simulators. Nevertheless, after more than 10 years of development, new software engineering problems have appeared: code

[1] IFP Energies nouvelles (IFPEN) is a French public research, innovation and training center active in the fields of energy, transport and environment. Cf. `http://www.ifpenergiesnouvelles.com`

duplication, design re-factoring issues, and difficulties in switching from specifications to code due to technological complexity.

In recent years, we have seen the emergence of specific HPC training (such as those provided by Prace[2]) and more and more hires of computer engineers specializing in parallelism, in numerical simulation code development teams. However, in traditional companies with large numerical simulations teams such as CEA, code development is usually done by mathematicians and physicists. These are the Domain Specific Language *users* in the context of this paper. Most of them have PhDs in applied mathematics. They are experienced in code development in FORTRAN and sometimes in C++. They naturally formalize the specifications in a Latex document. It is estimated that less than 1% of researchers have strong expertise in the programming of massively parallel architectures [2]. Then, they face a significant amount of concepts and technologies to design the physics module they have specified: object-oriented paradigm, C++ complexity, parallelism and integration of their development into large numerical simulation codes. As a result, there may be some time between the end of the specification process and the final source code due to the complexity of the technology stack.

Moreover, it becomes difficult to adapt complex numerical simulators to continuous hardware changes. In such a context, abstracting the design of simulation codes in order to generate parallel code for different hardware platforms appears as an attractive approach [3].

In 2012, CEA decided to explore the use of Model-Driven Engineering (MDE) [4] to develop Domain-Specific Languages (DSLs) [5] and dedicated design tools [6]. This led to two specific projects, namely Modane and NabLab. Modane makes it possible to graphically describe the structural and architectural part of a simulation code: the components (*e.g.,* variables, modules, services and structures) and the relationships between them (implementation, inheritance and composition). It also generates part of the source code of the simulator based on the Arcane framework. NabLab [7] allows the behavioral part of a numerical simulation code to be textually described by providing a modeling environment for Nabla[3], an already existing home-made DSL for numerical analysis algorithms.

In this article, we report on our experience through the Modane and NabLab projects. We first explain the initial objectives for adopting MDE and DSLs, and then the lessons learned. We also present the remaining challenges by proposing research tracks in the field.

The rest of the paper is structured as follows. Section 2 gives the details of the main initial objectives for conducting the two projects Modane and NabLab, while Section 3 describes the actual developments that have been done on these two projects and representative use cases . Then we discuss in Section 4 the main lessons learned, and in Section 5 the remaining challenges to be addressed in the future. Section 6 reflects on existing and related work to position our contribution, and Section 7 concludes the paper.

[2]Partnership for Advanced Computing in Europe. Cf. `http://www.training.prace-ri.eu/`

[3]The Nabla domain-specific language provides an efficient development mode for exascale computing technologies. The Nabla optimizing compiler translates numerical analysis algorithms, i.e. Nabla source code, and generates optimized code for different targets and architectures. Cf. `http://nabla-lang.org/`

2. Main Initial Objectives

In this section, we present the main objectives for initiating the Modane and NabLab projects, and describe our initial expectations from MDE and DSLs.

Capture and perpetuate domain knowledge. Most of the time, the knowledge in our simulators is spread across the specification documents and the source code. Very few design documents describe the global architecture of the key domain concepts within the software despite the complexity of the domain and the large number of technologies used, especially C++, parallelism (MPI for domain decomposition/replication and multi-threading) and embedded frameworks and libraries. Over time, the source code becomes the reference in which knowledge and useful information are lost, for example in C++ classes, mixing technical concerns and domain concepts. It becomes crucial to separate these two concerns since, in the HPC community, hardware generally evolves faster than complex simulation codes (*e.g.*, a new generation of cluster arrives every 5 years whereas it takes about 10 years to get an operational code).

One of the main interests of using MDE and DSLs is the ability to capture key domain concepts in a steady form, while code generators take into account technical concerns to provide efficient source code and bridge the gap between key domain concepts and their implementation in simulation codes.

Support of communication and development. UML class diagrams have been conventionally used to brainstorm and collaborate on the overall architecture of the application, relying on graphical representations. Unfortunately, these models were established independently and, as a result, they were quickly out of sync with the code and no longer useful.

Design models built with Modane and NabLab drive the generation of the corresponding code by generative approach, in order to centralize the brainstorming and development activities on a single artifact.

Improve software quality. Recent code reviews of our HPC codes highlight some design flaws. First, interfaces are underutilized. Since interface concept does not exist in C++, developers often share code in top-level classes that should be interfaces. Second, classes contain too many lines of code because creating a new class means creating header and body files and managing dependencies. In addition, classes are strongly interdependent due to flawed technical rationale, masking real useful domain dependencies. Last but not least, inheritance is misused for any code sharing, involving deep irrelevant inheritance trees instead of using well-know design patterns (*e.g.*, delegation).

By reifying relevant abstractions in domain models (i.e., metamodels) of DSLs and providing the associated generators, we prevent the frequently encountered anti-patterns in our simulation codes. In addition, conformance checking ensures that the model respects domain-specific structural constraints (aka. context conditions) and naming conventions. Finally, we rely on code generators to provide a uniform source code, thus ensuring homogeneity, compliance with software coding rules, and ultimately improving comprehensibility.

Table 1: NabLab and Modane metrics

#	Modane	NabLab
Meta-classes	40	49
LOC (Xtend & Java)	11494	5269
Manpower	≈ 45 man/month	≈ 10 man/month
Projects	3 industrial	1 research (mini-app)
Users	≈ 20	1 pilot user
Model size	≈ 17500 elements (for all models of the 3 projects)	≈ 750 elements on 160 NabLab lines (for 1 mini-app)
Total Applicative LOC	909273 (xml & C++)	286 (nabla file) -> 1600 (C)
Generated Applicative LOC	303697 (xml & C++)	286 (nabla file) -> 1600 (C)

Out of the scope of the initial expectations. To obtain the aforementioned benefits, the MDE has been from the beginning considered as a good candidate. However, at the outset, graphical modeling has been considered irrelevant to deal with very detailed and low level behaviors in simulation code. Our experience with *Unified Modeling Language* (UML) shows us that trying to capture such algorithms (*i.e.*, fine-grained mathematical operations) in graphical diagrams (*e.g.*, sequence or activity) is unsuitable. Therefore, to cover the behavioral part of codes, the Nabla language project started based on quite different technologies (*i.e.,* C, Flex, Bison). When Xtext[4] and textual modeling first appeared, we launched the NabLab project to provide a textual modeling environment for Nabla programmers.

3. Description of Developments

In this section, we present the two projects based on the *Eclipse Modeling Framework*[5] (EMF) [8] and its ecosystem. The first, namely Modane, is a modeling environment used since 2012 by 20 mathematicians and physicists on 3 industrial projects (cf. Table 1) to design and generate the architecture of their numerical simulators. The second, NabLab, is recently deployed, and is currently being used on a research project with a pilot user. Its purpose is to provide a modeling environment for the Nabla language, a domain-specific language for numerical analysis algorithms.

3.1. Modane

3.1.1. Development of the modeling environment

Modane is a modeling environment that deals with the architectural aspect of simulation codes [9]. The product includes a graphical modeling environment and code generators.

Modane users are the engineers in charge of the development of HPC numerical codes. The modeling environment allows them to design their simulation code. Users can graphically create components according to the global architecture specifications (Fig.

[4]Cf. `https://www.eclipse.org/Xtext`
[5]Cf. `https://www.eclipse.org/modeling/emf`

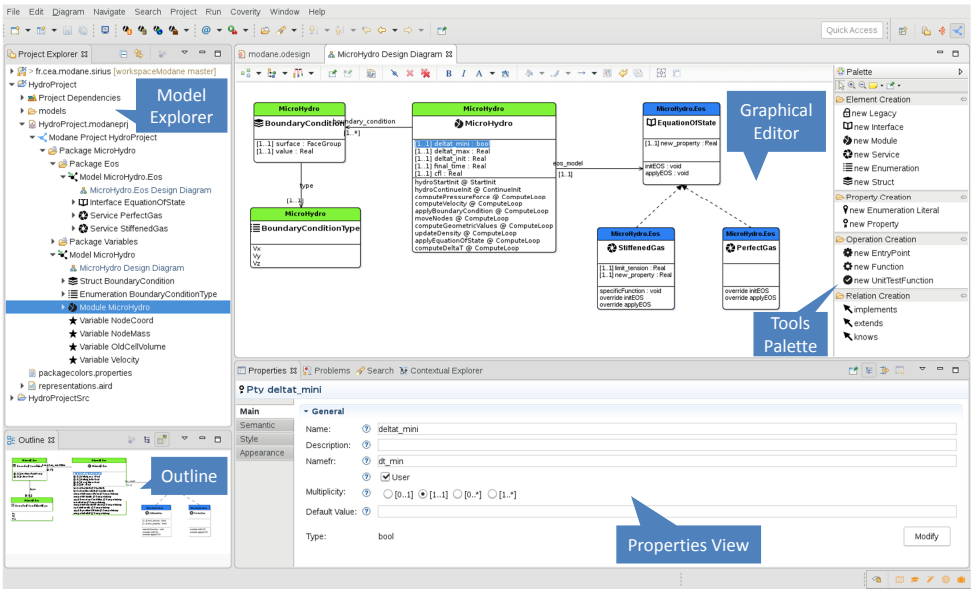

Figure 1: Modane Environment

1, palette and properties view). They can then navigate in the model (Fig. 1, model explorer) and create graphical diagrams for specific purpose (Fig. 1, graphical editor). They can also check a set of validation rules on models.

The modeling environment provides about 40 numerical simulation concepts (cf. Table 1): physical modules, services, mesh components such as cells, nodes or faces, and basic types. It also contains variables representing physical quantities, such as temperature or pressure.

Code generators produce both XML component descriptor files for Arcane and C++ code (complete base classes and final classes skeletons). The methods' bodies must then be completed by the users. When it is necessary to regenerate code after a modification of the model, we follow the *Generation Gap* pattern [10, Chap. 57] as a pessimistic approach: files completed by users are never overwritten (generated only once). To ensure consistency, user-edited classes inherit from fully generated classes.

The combination of all the design models for our 3 complex simulators brings together about 17000 elements. Approximately 300 KLOC of C++ are generated which represents 33% of the total LOC after users have completed the operations' bodies (cf. Table 1).

Modane is written using Ecore[6] and Sirius[7]. The hand-written source code consists of approximately 10 KLOC of Xtend[8] (cf. Table 1), including checking and validation functions, classes for menu handlers, code generators, and services used by the Sirius representation. Although Modane is now fully Eclipse EMF based software, this has

[6]Ecore is the meta-language provided by EMF to design a metamodel as an object-oriented domain model

[7]Cf. `https://www.eclipse.org/sirius`

[8]Cf. `https://www.eclipse.org/xtend`

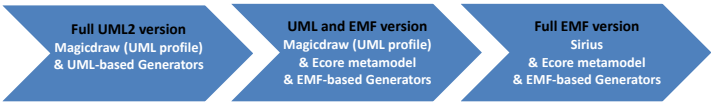

Figure 2: Modane Release History

not always been the case. We present in the following paragraph the 3 main successive versions of the product (Fig. 2).

UML Profile. The first version comes after 2 years of experimenting the use of UML for the development of new physical modules. A UML profile is then created to fill in domain concepts. The UML modeler used is Magicdraw[9] and Eclipse CDT[10] is our C++ development environment. The profile is thus integrated into Magicdraw and its customization capabilities are used to create a dedicated graphical modeling tool. Code generators are developed with Acceleo[11] to generate C++ source code from the UML2 models built with Magicdraw.

This version has two main drawbacks. From the point of view of the users, the tooling is heterogeneous (modeling in Magicdraw, C++ development in Eclipse), and for Modane development team, code generators are tricky to understand and maintain because they are based on a UML2 metamodel that contains too much information beyond the scope of the domain.

UML Profile and Ecore Metamodel. For the second version, an Ecore metamodel is created and the existing UML models are exported from Magicdraw and converted to conform to the metamodel. This Ecore metamodel is the support for code generators implemented with Xtend. The UML profile and the metamodel are aligned. In the case of metamodel and UML Profile changes, a Magicdraw converter must be implemented using the Magicdraw Java API, and the UML to Ecore converter must be updated. The overall architecture of this version is presented in Fig. 3.

With the Ecore metamodel, it is easier to write, understand and maintain code generators. However, the environment remains heterogeneous because users design their model in Magicdraw and then have to switch to Eclipse to write C++ code.

Ecore Metamodel. The current version provides a full-fledged environment based on EMF (cf. Fig. 1). Sirius is used to implement the graphical modeling environment (diagrams, tables and dedicated property views) over the Ecore metamodel. Modeling projects are now hosted in Eclipse nearby code projects. Existing EMF models resulting from the conversion of Magicdraw models have been imported.

This version offers an integrated model-to-code environment, which is appreciated by Modane users who are also code developers. While this sounds quite promising from the point of view of innovation potential, it does present some degree of instability because of the rapid evolution of the EMF ecosystem. This requires more direct involvement with the community to resolve bugs and complex situations, and to monitor upgrading

[9] Cf. `https://www.nomagic.com/products/magicdraw`
[10] Cf. `https://www.eclipse.org/cdt`
[11] Cf. `https://www.eclipse.org/acceleo`

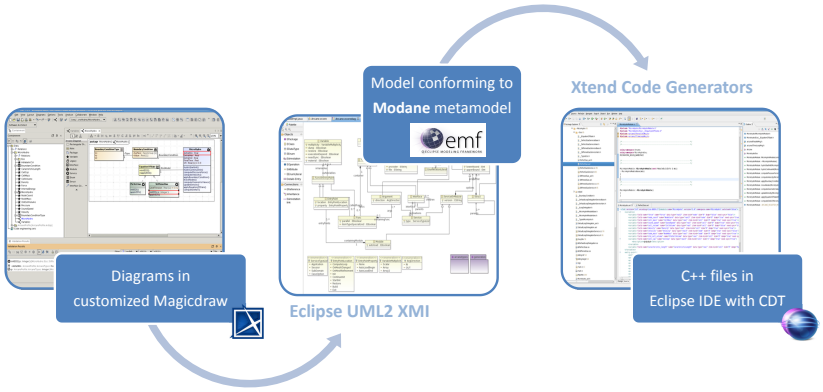

Figure 3: Modane workflow with an UML Profile and an Ecore metamodel

versions of the framework. The issue of metamodel evolution is still an open challenge (see Section 5).

Agile practices are now well accepted in the software development. Some of them were applied in the context of the Modane project, in particular pair-programming during the design and development of the project. It provided real-time review, and globally raised the level of quality eventually reached (e.g., less bug fixes).

3.1.2. Context of use in HPC and representative use cases

The Arcane framework is the foundation of several numerical simulation codes at CEA. These codes are designed to model complex physical phenomena, converting them into equations using numerical schemes. The domain is discretized as a mesh, composed of nodes and cells, in two or three dimensions. The physical quantities are the variables of the mathematical equations, they are calculated for each element of the mesh, for example the temperature or the pressure for each cell of the mesh. Time is also sampled as time-steps and the equations are solved at each iteration before going to next one.

Numerical simulation codes are developed by teams of a dozen developers. Simulation codes comprise tens of thousands of lines of C++ code and their life-cycle is about twenty years. This kind of complex multiphysics codes must be executed on supercomputers to obtain reasonable execution time. They must integrate several models of parallelism in order to improve the restitution time of the results.

Modane is used by mathematicians and physicists to brainstorm through the graphic representation of their models. Then they can agree on the structure of the code, and then define the physical modules, the options of the user data-sets, the manipulated variables, the functions to implement with their arguments and the variables concerned.

Once the model is validated, code generation can be launched. Modane users can fill in the methods code to implement. We follow a model-driven approach. When a change to the code structure is needed, users modify the model and generate the code again. As a result, the model is always up-to-date and can be used as a means of communication and training for new collaborators.

For the moment, no reverse engineering mechanism is available to postpone C++

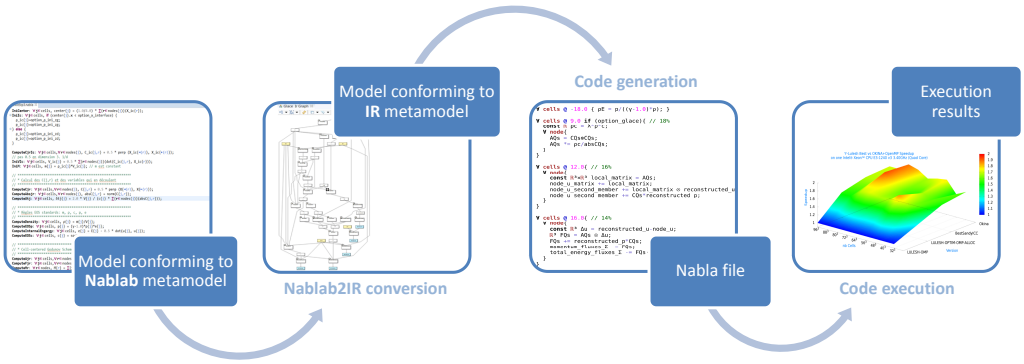

Figure 4: NabLab workflow

code changes on the Modane model. Such a feature requires annotating C++ code to maintain synchronization with the model and our code developers want to keep their code as "pure" as possible. We will probably have to offer them solutions in the future.

Modane is not intended to address the issue of code performance and parallelism, except for small parts such as, for instance, data parallelism. Parallel loops are generated for mesh items when the user states in the model that a function is to apply in a parallel way on a set of nodes or cells. The generated C++ code is based on Arcane parallel API. However by improving the modularity of the code, Modane makes it possible to identify time-consuming components. These ones can easily be pinned down and optimized, which contributes to the durability of the developments in the specific context of HPC.

3.2. NabLab

3.2.1. Development of the modeling environment

NabLab is a modeling environment that deals with the behavioral aspect of simulators, i.e., numerical analysis algorithms [7]. NabLab is the modeling environment for the Nabla Language and provides a compilation chain for HPC. Unlike the code generated by Modane, the code resulting from the NabLab workflow does not need to be completed by users and is directly executable (cf. Fig. 4).

NabLab environment provides a textual editor and an advanced environment for debugging and tuning applications (cf. Fig. 5). The textual editor, based on Xtext, proposes contextual code completion, code folding, syntax highlighting, error detection, quick fixes, variable scoping, and type checking. As shown in Fig. 5, the tool also proposes a model explorer and a dedicated outline view (left), as well as a LateX visualization of the selection in the editor (bottom). The debugging environment, still under construction, will provide at least variables inspection, curve display and 2D/3D visualization. Finally, as a NabLab program is composed of jobs, its execution does not start sequentially at the beginning of the program but a data flow graph is computed and displayed thanks to a Sirius diagram (cf. Fig. 5 dataflow graph). It shows the execution flow of the program and highlights possible cycles. From there, a Nabla program is generated and compiled.

NabLab corresponds to 5 KLOC of hand-written Xtend code (cf. Table 1), including all editor features (validation, scoping, typing, quick-fixes, LateX and Nabla generation)

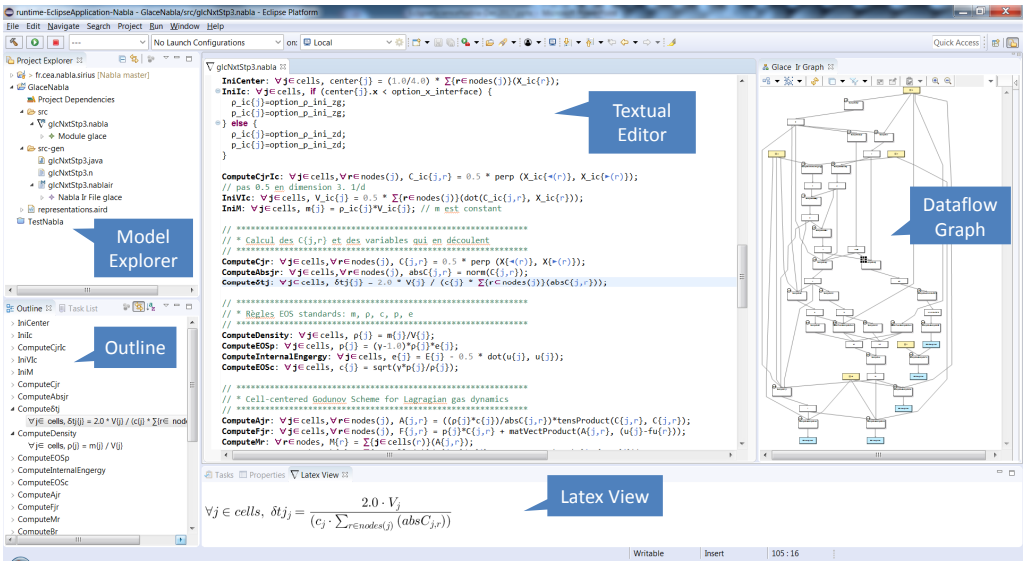

Figure 5: NabLab Integrated Development Environment

but also the graph construction. It works like a compiler chain: the NabLab model is transformed and enriched with execution flow information to an intermediate representation (IR) defined by its own metamodel. NabLab code generators are based on this IR metamodel. Fig. 4 shows the workflow starting from the textual editor to the IR and up to the generated code. Most NabLab design patterns and principles come from L. Bettini's Xtext book[11].

The NabLab project started during the summer of 2016 and is still under development. Only pilot users have used both the language and its environment on a mini-application of about 160 LOC which represents about 750 model elements. The Nabla compiler produces 1600 LOC of high performance C, 10 times more than the model LOC (cf. Table 1).

3.2.2. Context of use in HPC and representative use cases

NabLab is a prototype on which only a research application modeling the dynamics of Lagrangian gases in two dimensions has been implemented. Both a mathematician and a computer scientist worked on this model, and results are compared to the execution results already known.

The NabLab syntax allows to describe the mesh connectivity, and the variables for each element of the mesh. Each job describes a unitary treatment. The DSL allows users to define in a syntax that they easily understand, the instructions for each job, including loops on mesh items which represents the major part of the computational work. The data flow graph is then used to show the partial order of jobs and helps users to validate their algorithm.

Unlike Modane's generation mechanism, the Nabla source code is 100% generated. Therefore, it is necessarily a top-down approach (i.e., model to code) and no round trip is planned. Nevertheless, information is conveyed during the transformations on the intermediate representation so as to be able to link the DSL instructions and the

generated code lines. Up to now, the focus is about the checking of the generated code but, at last, we intend to debug the code directly from the model so the user does not have to deal with the generated code.

From an HPC point of view, NabLab supports both data parallelism (loops on mesh items) and jobs parallelism. The computed data flow graph allows users to identify jobs that can be executed in parallel. Thanks to these mechanisms, the Nabla code obtains sustainable run-time performance notably demonstrated on LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [12].

4. Lessons Learned

In this section, we discuss the main lessons learned from the implementation of the two projects introduced in the previous section.

4.1. Benefits of domain modeling

DSLs and associated code generators have reduced the gap between the mathematicians domain and the implementation code. Dedicated common concepts facilitate the dialog between computer scientists and mathematicians and make it possible to clearly define the boundaries of their respective fields. Although it is difficult to compare different projects, the time to deliver the initial version of the first Modane based simulator, was significantly shorter than in a previous comparable classical development process.

Before using DSLs, we tried to introduce the use of UML during the specification process to isolate domain knowledge. Despite this work, we did not succeed in totally avoiding mixing technical classes and domain classes in the C++ implementation. The use of DSLs enables us to keep a domain-specific design model.

> **L1** By establishing a clear separation of the concerns between physics, mathematics and computer science, DSLs improve the collaboration of such stakeholders leaning on the domain model.

Our users have quickly adopted the model-driven process that implies modifying the model to upgrade the code, regarding to the benefits of an up-to-date and consistent model of the simulator. The graphical representation facilitates the understanding of the overall architecture of the software, and the integration of new collaborators. Moreover, code re-factoring is easier and users do not hesitate to upgrade the design of the overall architecture.

A survey of our project managers illustrates this. Before the use of Modane, newcomers were assigned to limited developments on isolated small parts of the code before embracing the overall project. With Modane, newcomers start with a localized development for training purpose, but then they are directly affected on a large development together with the rest of the team. Hence, the project manager moved from 6 * 1 month of small development to 1.5 months allocated to newcomers. The integration time has been divided by 4. The project managers also quantified the time to re-factor the code for large architectural changes based on a restructuring experience at the beginning of the project when unidentified pooling opportunities were found to be possible. This big restructuring robbed the teams for a month, and the project managers estimate that it would have taken 2 to 3 times more without Modane.

As shown in Table 1, the code generator of Modane provides more than 300 KLOC of code from a set of models that gather 1700 elements. These figures are representative since they cover the whole 3 industrial projects using Modane. In order to properly design the 3 simulators, it is obvious that handling 1700 model elements is easier than 300 KLOC. The additional 600 KLOC of hand-written code on these 3 projects are mostly made of the dynamic part of algorithms that is not filled in the model.

We expect to bridge this gap in the near future thanks to the coupling of Modane and NabLab. At the present date, only a pilot project is using NabLab. This project represents a hydrodynamic module whose NabLab description is very compact (only 160 lines). The code generator provides the complete applicative code, composed of about 1600 LOC in the C language.

L2 The use of domain concepts eases the communication between stakeholders and the transfer of skills to new collaborators.

4.2. Concepts and metamodel

Our experiments highlighted the importance of keeping a sustainable domain model (i.e., metamodel), in particular to avoid major evolution of generators and graphical representations. Therefore, the domain model must be composed of concepts resulting from a good domain analysis, going beyond the specific needs of the users at a given moment. The main part of our metamodel is based on time-honored domain-specific concepts, all experienced within Arcane used for 10 years when Modane started. We took all domain concepts including those that were not used in a first step.

To date, the only time we have incorporated a non-domain concept in order to facilitate the development of a tool, we realized that this concept was quickly becoming obsolete (less than one year).

Very few changes have occurred on the metamodel since Modane is in production. We have identified 16 modifications on the metamodel during 7 years of product life. About two per year since 2013, most of which are very small additions. And each of these evolution is independent of tools but linked to an evolution of knowledge. The few changes did not limit the development of tools.

The stability of the metamodel is all the more important as the conversion of a model from one version to another of the metamodel remains complex.

L3 Considering time-honored concepts in the domain model (i.e., metamodel) favors its stability, independently of the current and specific needs of users.

As mentioned in Section 3.1, code generators in Modane were initially experimented from UML models. Despite the use of a profile dedicated to numerical simulation concepts, we have found no way to reduce UML expressiveness. Consequently, information could be flooded in UML concepts. Moreover, because the UML metamodel is "fine-grained", it requires a very good understanding to implement validation rules and code generators.

Here are some numbers to illustrate this complexity. The Ecore metamodel representing UML2 contains 256 concepts (EClassifiers). Models exported from Magicdraw contain objects of only 23 different classes among these concepts. It can therefore be estimated that 90% of UML concepts are not used.

Another metric is the size of the handled files, correlated to the number of model element . To represent a given domain model, as an instance of UML.ecore, the set of files

represents 11MB including 5MB for the model itself + 3MB for Modane profile + 3MB for UML. The same domain model, as an instance of modane.ecore weighs only 2.9 MB.

Finally, code generators based on UML are trickier to maintain. As stereotypes are referenced by a string identifier in UML2 model API, a change in a profile does not ever imply an error in the code.

L4 UML Profiles might lead to complex generators and expressiveness issues when the targeted domain deviates from UML.

4.3. Code Generation

To enhance acceptability, we proposed code generators that keep the code under control of the users. Indeed, we initially provided generators assuming that our users fill their code in source files between `begin-user-code` and `end-user-code` tags. This mechanism is not well-accepted by our users because it can rearrange the contents of the file.

It should be noted that the work organization in the simulator development team consists of defining a manager per part of the code. Each developer is therefore in charge of a set of files, which makes him individually responsible and pushes him to wish a total control of his files.

Hence, the current C++ generator follows the *Generation Gap* pattern[12], which keeps the generated code separate from the handwritten code through inheritance mechanism. It results in a systematic code design and a readable code architecture.

Tag-based code generation, however, offered a best generation-code ratio, in terms of code to be written by the language users. For example, in the case of a modification of the signature of a method, the generation performed the entire work, whereas the current generators require the end-user to rewrite the signature of the method in the C++ body. Despite this, our users prefer this new version.

L5 Users want to keep control over their own code when integrated with generated code (*e.g., Generation Gap* pattern).

Some of our users may not be used to C++ because they are FORTRAN developers and have recently been integrated into the development of the simulator. As a result, they use the generated code as a template, they often copy/paste it. Therefore each pattern must be highly considered and a particular attention must be paid when writing the generation templates. The resulting code should be in line with the best development practices of the moment. As the code will be duplicated in the rest of the simulator, it must follow the latest C++ evolution (C++ 14, then 17) and have no warning on the strictest compilers (so it must be tested by several compilers).

L6 The generated code must be well-crafted as it will be considered as a reference and copied to be later adapted

Most of our simulators are based on the Arcane framework. Over the time, it is difficult to evolve the Arcane API, because it requires to change calls in the code. Model-driven

[12] Cf. `https://martinfowler.com/dslCatalog/generationGap.html`

developments make it possible to reinvigorate the addition of features in the framework since it only requires to evolve the generators to apply the change in the whole code base.

L7 Generative approaches favors the evolution of the targeted platforms.

4.4. Tools

Modane is a very innovative product in the CEA ecosystem. Users are mathematicians who turn out to be code developers. By introducing modeling languages in the development of numerical simulators, we have changed their coding habits.

The model-driven approach imposes to modify the model to regenerate the code even for a minor modification (adding a parameter or changing the cardinality of a data). In this context, some users may be tempted to leave their modeling environment to save time. Similarly, some users with advanced knowledge of C++ are frustrated at not being able to use tricky C++ mechanisms (such as templates, lambdas) and having to stay in the straitjacket imposed by generation.

The interest of an always up-to-date model is obviously perceived by managers (graphical view of simulator architecture and training of new developers). To convince users of the efficiency of this approach, it is necessary to propose them some high level functionalities which bring them a real advantage compared to the development in a traditional IDE. For example, we have developed a cross-dependency search tool that graphically presents dependency cycles between components and allows the code developer to reduce couplings between the software parts.

More generally, we believe that for a DSL-based environment to be adapted and accepted by users, it must have the following non-exhaustive characteristics: relevant language, diagrams, validation and verification tools, operability (non redundant information).

L8 A rich modeling environment is required to convince code developers to follow a model-driven approach.

Often, users develop specific tools around simulators. For instance, to extract information from the code to build reports. The implementation of these tools depends on the user: Python, Perl, shell.

The level of abstraction offered by EMF enables users to simply develop their own dedicated tools. For example, we suggested a resourceful user to use the EMF-generated Java API to query the model to develop generators for specific documentation and to build a GUI for entering the code data-set. Previously, he was trying to make his tool using the Magicdraw export with Python tools and xslt queries to dynamically search information in the xmi serialized model. It takes him a little time to get used to the metamodel (very close to his domain) and the Xtend language, but he quickly gained in efficiency by working directly on the model instead of using untyped xpath queries.

The emergence of this new kind of tools appears like an experimental laboratory and some of them should be integrated in Modane when they will be mature enough.

L9 The domain model (i.e., metamodel) can be used directly by resourceful end-users to develop their own dedicated, usually small, facilities (*e.g.,* analysis and generation tools).

Table 2: Generation time metrics

Number of model elements	Generation time	Number of generated files	Number of generated loc	Compilation time of generated code
17 675	1 minute	4 046	303 697	10 minutes

The performance of the tools in the modeling environment is better than initially expected, and in any case acceptable by our users. For example, Table 2 gives the generation time on the largest model of our projects. It should be noted that code generation is always triggered partially on a small part of the model on which the changes took place (incremental generation) and then, the generation time is reduced to a few seconds (negligible in the context of an optimizing compiler).

L10 Overall performances of DSL implementations (e.g., compilers and analysis tools) are acceptable.

The development team appreciates the fast evolving EMF ecosystem, always providing new cutting-edge technologies, but it requires continuous technology watch.

The first version of Modane was based on MagicDraw. This off-the-shelf tool proved to be very efficient and particularly ergonomic. Deploying this version of Modane was simpler than deploying the fully integrated version of Eclipse, which raises for each version, the problem of compatibility of the various Eclipse plugins that make it up. However, the Eclipse version offers us much more advanced integration possibilities thanks to graphical API and plugins.

Our learning curve on EMF was quite exponential: getting a prototype is very fast (only a few weeks), but getting an industrial version, that deals with tricky concepts such as scope providers, can eventually become expensive (cf. Table 1). This is all the more true given that we integrate several tools as Xtext and Sirius.

Currently, we try to keep our products in touch with new Eclipse versions around two times a year. From this point of view, Eclipse open-source ecosystem is a powerful engine for innovation. Indeed, one of the advantages of using an open-source development environment with an active community is to offer new opportunities continuously. Thanks to this, we have been able to develop new features during the project, based on tools that did not even exist at the beginning.

As an example, we have developed a view showing the dependencies between model objects with 3 parts: in the middle, the object selected in the editor, on the left the elements on which our object depends and on the right those who depend on it. This view is based on the Contextual Explorer, part of Capella project. Capella was created by Thalès in 2007 and released as an Eclipse open source project in 2015. This use of the Contextual Explorer does not answer an expressed need but has been identified as an interesting improvement during a Capella demonstration and today our users greatly appreciate it.

This example shows how this kind of cutting-edge technology promotes open innovation, provided that a sufficient technological watch is maintained on the new products.

L11 EMF ecosystem brings capabilities for innovation as long as you ensure a regular technological watch.

5. Open Challenges

Modane is fully deployed and used on a daily basis by dozens of users. NabLab is still under development and needs many improvements to be fully integrated in the environment. To date, several challenges remain open and will require further investigation in the future. In this section, we review them through a classification that brings several big challenges to the community.

Socio-technical coordination. Technical coordination, i.e. automated processing of possible interactions between various heterogeneous models, is one of the challenges ahead [13]. Although most of the language users are homogeneous (i.e., mathematicians), technical coordination is required beyond the scenarios of collaborative modeling (consistency, synchronization). Synchronization and coordination of textual and graphical views are also part of the current limitations. The support of the communication between mathematicians and computer scientists is also challenging. Indeed, although the use of DSLs enables to dissociate the concerns of language users and designers (i.e., mathematicians and computer scientists), it is also very important to keep a communication channel between them, both to ensure the proper development of the DSL and to foster the identification of future relevant features.

Cartography and review of the existing tools in the different ecosystems supporting MDE and DSLs. In the context of NabLab, we essentially focused on a convivial environment to edit the model: textual editor with auto-completion, syntax highlighting and code folding; dedicated view with LateX images for mathematical formula; graphical representation of the underlying data flow. All these functionalities are basically offered by common tools such as Xtext and Sirius. However, the definition of advanced tools such as debugger, variable inspector, execution visualization (e.g., plotting or 2D/3D visualization) are not covered by such tools. While additional tools are proposed in the literature to address part of these facilities (such as *Nebula*[13] or ICE[14]), it is still difficult to get a consolidated view of what exists, including pros and cons of the different approaches.

Build and deployment automation (DevOps). While recent language workbenches support most of the tasks to develop a DSL, there is still a lack of support to automate integration, tests and release. While Git and JUnit are already used in our two projects, tools supporting the DevOps principles are not yet ready (*e.g.*, continuous integration, delivery and deployment), and are required to improve support for the iterative and agile language development life-cycle.

Metamodel forces and frictions. Metamodeling aims at defining the essential abstractions suited for manipulating a set of models. The definition of a metamodel is a difficult yet crucial activity toward the development of specialized tools (*e.g.*, editors, visualizers, checkers). Although, in our context we experienced fairly stable metamodels thanks to proven concepts from previous experiments, many additional friction forces influence the metamodeling process, e.g. the technology used, or the services to be implemented on top of the metamodel. While it is pretty easy to start from a *clean* metamodel that best represents the domain, there is no particular support to specialize it according to the needs coming from specific service implementations (e.g. bidirectional reference

[13] Cf. `https://www.eclipse.org/nebula/`

or alternative data structure to best support the implementation of the service). To date, either we complement the metamodel with the risk to spoil it or to face conflicting needs, or we build a new metamodel specifically for the service to be implemented and provide a bi-directional transformation between the two metamodels. As bi-directional transformations are difficult to master in the general case, it would be helpful to provide a particular support for simpler cases, e.g. when one metamodel is an extension of the other one.

Metamodel and model co-evolution. When comes the time to improve the metamodel with regards to a better understanding of the domain, facilities for the co-evolution of existing models are of uttermost importance. To date, no automatic solution exists for this need, and a specific model transformation has to be implemented each time a change appears in the metamodel. There is thus an urgent need for an approach that could monitor changes on the metamodel, and automatically generate a patch to upgrade all models.

Concurrent work. DSLs improve collaboration between physicists, mathematicians and computer scientists in providing a clear separation of concerns (Lesson L1). However, thanks to Modane, people from the same domain, especially mathematicians, want now to work concurrently on the same model. Initially, we decomposed the model to avoid conflict. With the increasing number of users, the technical divisions have overshadowed the domain divisions. To avoid this, our main challenge now is to support the merging of separately edited models. So, we are currently investigating the use of EMF Compare[14] reconciliation rules.

Technological space independence. EMF ecosystem brings capabilities for innovation (Lesson L11) and it is well appreciated by DSL developers. However, this technological choice imposes that users work in an integrated Eclipse workbench. More generally, while language workbenches bring great facilities to implement DSLs, they are strongly dependent of their own technological spaces. To date, there is no *de facto* standard that would make language implementation less tied to the technological space of the language workbench used to implement it. Initiatives such as the *Language Server Protocol* (LSP) are going in this direction, and need to be extended to support interoperability between languages, modeling environments, and execution platforms. In the context of NabLab, this would enable interoperability with other development environments (e.g. CLion), or even to offer web-based services (e.g. editors) connected to cloud-based services (e.g. workspace) and server-side services (e.g. simulator).

6. Related Work

The use of DSLs for HPC has been intensively explored and reported in the literature since a long time, especially to address various specific concerns (*e.g.*, architecture [15], Stencil codes [16], deployment on parallel platforms [17, 18] or cloud platforms [19], etc.), or even specific application domains (*e.g.*, fluid dynamics [20], nuclear [21]). It has been the core topic of the WOLFHPC workshop series[15] at the SC conference[16]. More recently,

[14]cf. `http://www.eclipse.org/emf/compare`
[15]Cf. `http://hpc.pnl.gov/conf/wolfhpc`
[16]Cf. `Cf.http://www.supercomp.org`

problem-specific DSLs for HPC has been explored in the context of MDE, and has been the core topic of the MDHPCL workshop series[17] at the MODELS conference[18]. However, while there is an important literature on specific DSLs for various concerns and application domains related to HPC, there is very few literature on the actual lessons learned from this experience. In this paper, we focus on reporting the experience from two real-world DSL projects for HPC, incl. the design, development, deployment, and maintenance of the DSLs.

Several experience reports have already been published, *e.g.,* to propose design patterns for the use of DSLs [22], or to report the benefits of using DSLs [23]. Other experience reports have also already been published with general lessons learned on the design and implementation of DSLs. For instance, David Wile [24] presents the experience of three different DSLs and discusses several lessons learned. However, if some of the lessons learned overlap with those presented in this paper (e.g. regarding the elicitation of the targeted DSL domain), the author focuses on additional lessons learned from the perspective of deployment and adoption. In this paper, we focus instead on the lessons learned from the point of view of the language engineer, in the specific domain of HPC. Therefore, we develop more concrete lessons, but limited to the field of HPC. This allows us to provide actionable recommendations for future development in a similar context, and to raise concrete challenges for the language engineering community.

The most related work that reports on experiences in the context of HPC is certainly Ober at al. [25], which provides evidences on the usefulness of MDE for HPC. In a complementary way, we propose in this paper a deep review of our experience from a language engineering point of view, and provide both technical lessons leaned and future challenges for the MDE community.

To the best of our knowledge, we report for the first time in this paper, from a language engineering point of view, the lessons learned from two important and sustainable projects of DSLs for HPC, which exhibit an actionable research road-map for future work.

7. Conclusion and Perspectives

The development of Modane and NabLab, as well as their user acceptability, is a promising experiment in the use of MDE for HPC. The initial objectives listed in Section 2 have been met. The models made with Modane contain most of the domain knowledge of simulators. As noted in Section 4, mathematicians and physicists use the models on a daily basis to brainstorm, and newcomers use them to understand the architecture of the simulation code. Thanks to metamodel constraints, model validation rules and code generators, the quality of the final software has increased on 4 key points: low coupling, smaller classes, less inheritance and code consistency.

One of Modane's major assets is based on the very stable and proven concepts used for simulation code architectures. These concepts have been proven by many years of use of the Arcane framework at CEA. The result is a reliable metamodel that has hardly changed since the beginning of the project. This is a considerable asset as regards maintenance and co-evolution of existing models, in a context where our codes have quite long lifetimes.

[17] Cf. `https://www.irit.fr/MDHPCL2013/MDHPCL2014`
[18] Cf. `http://modelsconference.org`

The gap between hardware and software lifetime is a key issue of HPC. The clear separation of concerns offered by the use of MDE allows numerical simulators to continually adapt to supercomputers. The real challenge has been to provide innovative and advanced facilities to users accustomed to conventional technologies such as vi and FORTRAN. Note that regarding the generation of skeletons of operations in Modane, they refused to enter their source code between generation tags and forced us to use the Generation Gap pattern in order to maintain control of their source code. But they quickly understood the high potential offered by the level of abstraction of the models and by a range of adapted tools. A traceability model allows to trace back the execution errors at the model level, and would possibly support reverse engineering solutions in the future.

We are currently thereon migrating the classic historical tools provided with numerical software to small Xtend applications based on EMF Java API. In the next step, pilot users seem interested in assimilating these technologies to create new domain-specific applications themselves.

Our main ongoing challenge is to combine Modane and NabLab to describe both architectural and behavioral aspects of simulation codes. To achieve this goal we need to add new features to the NabLab environment, such as full-fledged debugging. Then, NabLab and Modane metamodels have to interact together in an integrated environment. Such a modeling tool-chain should meet our need to facilitate adaptation and evolution according to the continuous hardware evolution.

References

[1] G. Grospellier, B. Lelandais, The Arcane development framework, in: Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing, POOSC '09, ACM, 2009, pp. 4:1–4:11.

[2] E. Pérez-Wohlfeil, O. Torreno, L. J. Bellis, P. L. Fernandes, B. Leskosek, O. Trelles, Training bioinformaticians in high performance computing, Heliyon 4 (12) (2018) e01057. doi:https://doi.org/10.1016/j.heliyon.2018.e01057.
URL http://www.sciencedirect.com/science/article/pii/S2405844018329797

[3] M. Mernik, J. Heering, A. M. Sloane, When and how to develop domain-specific languages, ACM Comput. Surv. 37 (4) (2005) 316–344. doi:10.1145/1118890.1118892.
URL http://doi.acm.org/10.1145/1118890.1118892

[4] D. C. Schmidt, Guest editor's introduction: Model-driven engineering, Computer 39 (2) (2006) 25–31. doi:10.1109/MC.2006.58.
URL https://doi.org/10.1109/MC.2006.58

[5] A. van Deursen, P. Klint, J. Visser, Domain-specific languages: An annotated bibliography, SIGPLAN Not. 35 (6) (2000) 26–36. doi:10.1145/352029.352035.
URL http://doi.acm.org/10.1145/352029.352035

[6] B. Combemale, R. France, J.-M. Jézéquel, B. Rumpe, J. R. Steel, D. Vojtisek, Engineering Modeling Languages, Chapman and Hall/CRC, 2016.
URL http://mdebook.irisa.fr/

[7] B. Lelandais, M.-P. Oudot, B. Combemale, Fostering metamodels and grammars within a dedicated environment for HPC: the NabLab environment (tool demo), in: SLE 2018 - International Conference on Software Language Engineering, Boston, United States, 2018, pp. 1–9. doi:10.1145/3276604.3276620.
URL https://hal.inria.fr/hal-01910139

[8] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Addison-Wesley Professional, 2008.

[9] B. Lelandais, M.-P. Oudot, Modane: A design support tool for numerical simulation codes, Oil Gas Sci. Technol. - Rev. IFP Energies Nouvelles 71 (4) (2016) 57.

[10] M. Fowler, Domain Specific Languages, 1st Edition, Addison-Wesley Professional, 2010.

[11] L. Bettini, Implementing Domain Specific Languages with Xtext and Xtend - Second Edition, 2nd Edition, Packt Publishing, 2016.
[12] J.-S. Camier, Improving performance portability and exascale software productivity with the Nabla numerical programming language, in: Proceedings of the 3rd International Conference on Exascale Applications and Software, EASC '15, University of Edinburgh, 2015, pp. 126–131.
[13] B. Combemale, J. Deantoni, B. Baudry, R. France, J.-M. Jézéquel, J. Gray, Globalizing Modeling Languages, Computer (2014) 68–71.
URL http://hal.inria.fr/hal-00994551
[14] J. J. Billings, A. R. Bennett, J. H. Deyton, K. Gammeltoft, J. Graham, D. Gorin, H. Krishnan, M. Li, A. J. McCaskey, T. Patterson, R. Smith, G. R. Watson, A. Wojtowicz, The eclipse integrated computational environment, CoRR abs/1704.01398.
[15] B. A. Allan, R. Armstrong, D. E. Bernholdt, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. Mclnnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, S. Zhou, A component architecture for high-performance scientific computing, Int. J. High Perform. Comput. Appl. 20 (2) (2006) 163–202.
[16] R. Membarth, F. Hannig, J. Teich, H. Köstler, Towards domain-specific computing for stencil codes in hpc, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, 2012, pp. 1133–1138.
[17] B. Tekinerdogan, E. Arkin, Pardsl: a domain-specific language framework for supporting deployment of parallel algorithms, Software & Systems Modeling.
URL https://doi.org/10.1007/s10270-018-00705-w
[18] X. Zhen, S. Jizhou, Y. Ce, W. Huabei, M. Xiaojing, T. Shanjiang, A visual model-driven rapid development toolsuite for parallel applications, in: 2009 WRI World Congress on Computer Science and Information Engineering, Vol. 7, 2009, pp. 479–483. doi:10.1109/CSIE.2009.806.
[19] C. Bunch, N. Chohan, C. Krintz, K. Shams, Neptune: A domain specific language for deploying hpc software on cloud platforms, in: Proceedings of the 2Nd International Workshop on Scientific Cloud Computing, ScienceCloud '11, ACM, 2011, pp. 59–68.
[20] S. Macià, S. Mateo, P. J. Martínez-Ferrer, V. Beltran, D. Mira, E. Ayguadé, Saiph: Towards a dsl for high-performance computational fluid dynamics, in: Proceedings of the Real World Domain Specific Languages Workshop 2018, RWDSL2018, ACM, 2018.
[21] M. Palyart, D. Lugato, I. Ober, J.-M. Bruel, Improving Scalability and Maintenance of Software for High-Performance Scientific Computing by Combining MDE and Frameworks, in: ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MODELS), Vol. 6981 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 213–227.
URL https://hal.archives-ouvertes.fr/hal-00760410
[22] D. Spinellis, Notable design patterns for domain-specific languages, J. Syst. Softw. 56 (1) (2001) 91–99. doi:10.1016/S0164-1212(00)00089-3.
URL https://doi.org/10.1016/S0164-1212(00)00089-3
[23] T. Wegeler, F. Gutzeit, A. Destailleur, B. Dock, Evaluating the benefits of using domain-specific modeling languages: An experience report, in: Proceedings of the 2013 ACM Workshop on Domain-specific Modeling, DSM '13, ACM, New York, NY, USA, 2013, pp. 7–12. doi:10.1145/2541928.2541930.
URL http://doi.acm.org/10.1145/2541928.2541930
[24] D. Wile, Lessons learned from real dsl experiments, Science of Computer Programming 51 (3) (2004) 265 – 290. doi:https://doi.org/10.1016/j.scico.2003.12.006.
URL http://www.sciencedirect.com/science/article/pii/S0167642304000310
[25] I. Ober, M. Palyart, J.-M. Bruel, D. Lugato, On the use of models for high-performance scientific computing applications: an experience report, Software & Systems Modeling 17 (1) (2018) 319–342. doi:10.1007/s10270-016-0518-0.